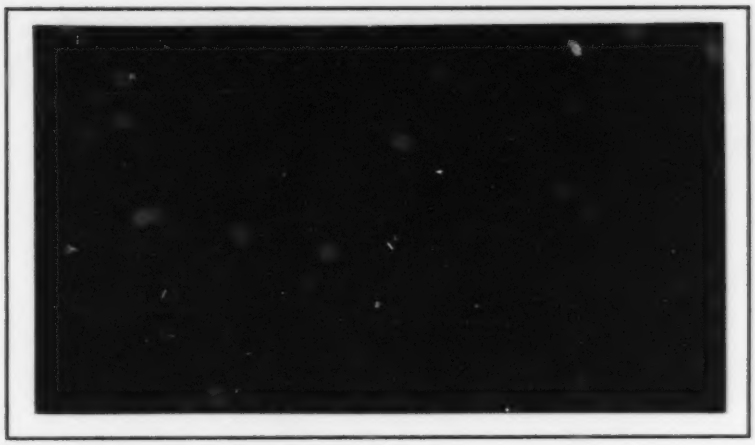

CENTRE FOR
SYSTEMS
SCIENCE

LCCR

Simon Fraser University, Burnaby, B.C., Canada V5A 1S6 / (604) 291-4704

Limitations of Process Management in Distributed Systems

F. Warren Burton
V.J. Rayward-Smith

CSS/LCCR TR90-04
CMPT TR 90-04

March 12, 1990

92-05792/

Limitations of Process Management in Distributed Systems[1]

March 12, 1990

by
F. Warren Burton[2]
V. J. Rayward-Smith[3]

[1]This work was supported by the Natural Science and Engineering Research Council of Canada, the United Kingdom Science and Engineering Research Council, the Simon Fraser University Centre for System Science, and the Tulsa Research Center of the Amoco Production Company.
[2]School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada V5A 1S6.
[3]School of Information Systems, University of East Anglia, Norwich NR4 7TJ, England. This work was done while Dr. Rayward-Smith was visiting Simon Fraser University.

Abstract

We will consider a system where processes may be created dynamically as a computation progresses and may synchronize with other processes in an arbitrary way. We will assume that a process cannot be moved once it has started.

We will show that the order in which processes are scheduled may make a spectacular difference in the performance of a program. In particular, we will show that for sufficiently large problems we can construct a program that can have a speed-up of m, where m is the number of processors, or a speed-up arbitrarily close to 1 (i.e. no effective parallelism) depending on the order in which processes are scheduled. We will argue that, in practice, a system is not likely to have sufficient information to be able to determine which process should be scheduled, so it is not possible to avoid bad schedules.

Even with complete information, finding an optimal schedule is shown to be NP-hard, even for tree structured problems and $m = 2$. Obvious heuristics also are shown to fail.

Finally, the problem of determining a good assignment of processes to processors is show to be impossible with the information that is likely to be available to a system at the time a decision must be made.

Our conclusion is that fully automatic scheduling of arbitrary parallel programs on systems that do not allow processes to move once started cannot be done in a way that will ensure good performance.

I. Introduction

Many proposed and existing languages for parallel computing support dynamic process creation and allow processes to synchronize in arbitrary ways.

For the moment, let us consider a deterministic parallel program running on a shared memory machine. We will assume that the work that is to be done by the program is fixed and that all results are uniquely defined. However the order in which things are to be done may be nondeterministic. In particular, we will assume that any processor may, at any time, run any runnable process. The program is a parallel program if there is more than one runnable process during at least part of the execution. If there are more runnable processes than processors, nondeterministic behavior is possible, even for a program with deterministic semantics, since there is a choice of which process to schedule.

If we ignore overheads and assume that the scheduler never allows a processor to be idle when there is a process that can run, then the worst possible schedule takes less than twice as long as the best possible schedule. In fact, the schedule length can be bounded in terms of the average parallelism in the problem (that is, the speed-up that would be possible with an unbounded number of processors), and the number of processors. If p is the average parallelism, T_1 is the time required by the program on one processor, T_m is the time required on m processors, and $S_m = T_1 / T_m$ is the speed-up on m processors, then Eager, Zahorjan and Lazowska [6] show that $\frac{mp}{(m+p-1)} \leq S_m \leq \min(p, m)$. If $m = p$ then $S_m > m/2$. If either m or p is very large compared to the other value, then the speed-up approaches the smaller of these two values and is always at least half the smaller value. For example, if the average parallelism in a program exceeds the number of processors, then a speed-up equal to at least half the number of processors is guaranteed, and the guaranteed speed-up must approach the number of processors as the average parallelism

becomes large. On the other hand, the speed-up cannot be greater than the minimum of the average parallelism and the number of processors. This result suggests that we do not need to worry much about scheduling.

While the choice of a schedule may make only a relatively minor difference in the time requirements of a parallel program, it may make spectacular differences in the space requirements [1]. If a program is nondeterministic, then any difference in the schedule can completely alter the behavior of a program. For example, in combinatorial search algorithms, such as Branch-and-bound, minor scheduling differences may make major differences in the time required by a program [2, 9]. Since we are demonstrating negative results, it does not matter that things can get even worse in a more general setting. We will limit our attention to deterministic parallel programs and consider only the time required to execute a program.

In this paper, we will consider what can happen in a distributed system. For our purposes, the one significant characteristic of a distributed system will be that once a process starts to execute on a given processor then it must remain on that processor. (In practice, a real distributed system may allow processes to migrate to other processors, but this can cause communication problems. For example, if a process uses a large data structure, and the process is moved frequently, then the communication cost of moving the data structure, or accessing the data structure remotely, may dominate the computational costs.)

In section II we will define our model, state our assumptions more precisely, and introduce terminology to be used in the remainder of the paper. In section III we will give an example of a program that can have a speed-up arbitrarily close to m, depending on the problem size, if scheduling is done in an optimal order, but will have a speed-up of arbitrarily close to 1 (i.e. no effective parallelism) if scheduling is done in the worst

possible order. We will further argue that it is not possible in practice to avoid bad schedules. In section IV we will show that computing the optimal schedule given complete information is NP-hard, even for very restricted problem classes. An obvious heuristic, level scheduling, which might be expected to give reasonable results with complete information and to suggest possible heuristics for practical systems, is shown to fail completely in section V. The problems discussed so far arise from the constraint that each process must be executed on a particular processor. In section VI we will briefly consider the problem of deciding which processor should execute a given process, and see that a system is not likely to have sufficient information to make a good choice. Section VII is the conclusion.

II. Model, Assumptions and Terminology

We will model a *deterministic distributed program on* m *processors* by an ordered triple, $(A, \subset, f)$, where A is a set of atomic actions, simply called actions hereafter, $\subset$ is a partial order defined on those actions, and $f: A \rightarrow \{1, 2, \ldots, m\}$ is a total function. If, for actions a_1 and a_2, $a_1 \subset a_2$, then a_1 must be performed before a_2. The function f is an assignment of actions to processors.

In the remainder of this paper we will assume that each action can be performed by a processor in one unit of time. This is not a serious restriction in our model, but simplifies the presentation. A process or task in a high level programming language corresponds to a collection of actions. A sequential process corresponds to a chain of actions (i. e. a collection in which for any two actions, a_1 and a_2, either $a_1 \subset a_2$ or $a_2 \subset a_1$).

A schedule is a total function $s : A \rightarrow \{1, 2, \ldots\}$, subject to the restrictions that

1) if $f(a_1) = f(a_2)$ and $s(a_1) = s(a_2)$ then $a_1 = a_2$, and

2) if $a_1 \subset a_2$, then $s(a_1) < s(a_2)$.

A schedule assigns actions to times when they are to be performed, subject to the restriction that no two actions are performed at the same time on the same processor, and the partial order $\subset$ is respected. If other than unit execution time actions are allowed, this definition must be generalized. We define the *length* of a schedule by $length(s, (A, \subset, f)) = \max \{s(a) \mid a \in A\}$. When $(A, \subset, f)$ is clear from context, we will write $length(s)$ for $length(s, (A, \subset, f))$.

An optimal schedule for $(A, \subset, f)$ is a schedule, s, such that $length(s) \leq length(t)$ for any other schedule, t, for $(A, \subset, f)$.

We call a schedule *work preserving* if a processor never idles when it could work. Formally, s is work preserving iff

3) $k \notin f(s^{-1}(t)) \;\&\; f(a) = k \;\&\; s(a) > t \Rightarrow \exists b(b \subset a \;\&\; s(b) \geq t)$.

That is, if processor k is idle at time t and action a is performed on processor k at some later time, then there must exist some action b that has not run prior to time t where action b must run before a. Clearly there is a work preserving schedule of optimal length so we will restrict all our discussion to such schedules. A schedule will be called valid only if it satisfies conditions 1), 2) and 3).

For the present, we will assume that the number of processes in a distributed program equals the number of processors, with one process per processor. Hence, we will use the term *process* to mean the set of actions assigned to a given processor. A process may contain internal parallelism. That is, the actions within a process do not need to be totally ordered with respect to $\subset$. If a reader likes to think of a process as a totally ordered set of actions (i.e. a sequential process) then he or she may wish to regard the collection of actions assigned to a given processor as a collection of processes that happen to be assigned to the same processor. In section VI we will consider the problem of assigning

processes to processors for the general case where there are more processes than processors.

In a practical system for distributed computing, a scheduler is likely to have only limited information. For example, if a system could determine how much time a process requires (i.e. how many actions the process contains), it could solve the halting problem. Therefore, a processor is unlikely to know how many actions are assigned to it, although it may know how many runnable actions it currently has. Similarly, if semaphore operations are contained in conditionals, then a system cannot predict how actions will synchronize. We will assume that, in general, a scheduler has no way to distinguish between runnable actions.

In job shop scheduling (e.g. see [4]) partially ordered UET sets of actions are assigned to processors. The basic model differs from our model in the sense that there is no predetermined assignment; any action can be processed on any machine. Jaffe [11] refined this basic model to include types of action and corresponding types of processor. He showed that with k types of action and m_i identical processors for actions of type i, the length, ω, of any work preserving schedule satisfies

$$\omega \leq [(k+1) - \frac{1}{\max\ \{m_1\ m_2\ \ldots\ m_k\}}] \times \omega_{opt}$$

where ω_{opt} is the length of the optimal schedule. Our model thus corresponds to a special case of Jaffe's model where $k = m$ and $m_1 = m_2 = \ldots = m_m = 1$.

III. Worst Case Scheduling

Consider the deterministic distributed program, which we will refer to as example 1, whose actions are illustrated in Figure 1a. Each subscripted variable represents an action. For actions a and b, $a \subset b$ iff there is a downwards directed path from a to b. The second

subscript of an action indicates the processor to which an action is assigned, i.e. $f(a_{i\,j}) = f(b_{i\,j}) = j$. There are m processors and also m chains of actions. Each chain contains k actions for some arbitrary (large) value k. The r^{th} chain ($1 \le r \le m$) comprises $a_{r\,1}, a_{r\,2}, \ldots, a_{r\,r}, b_{r+1\,r}, b_{r+2\,r}, \ldots, b_{k\,r}$. Notice that the first r actions are assigned to r different processors, while the remaining processes are all assigned to processor r.

This example is similar to one cited in [11]. In Figure 1b we give an optimal schedule for this program on m processors. Each row represents one processor, each column represents one unit of time, or a number of units of time if it contains an ellipsis, and the entry in a given row and column gives the action that the processor performs at that time, or is blank if the processor is idle. On the other hand, Figure 1c describes a poorly performing (depth first) schedule. While the schedule of Figure 1b is of length $k+m-1$, the poor schedule of Figure 1c is of length $mk-(m-1)(m-2)/2$. Since are mk actions, the first schedule has a speed-up tending to m, as $k \to \infty$, while the second has a speed-up tending to 1.

IV. Optimal Scheduling

In this section, we establish the result that finding an optimal schedule for an arbitrary deterministic distributed program on m processors, $(A, \subset, f)$, is NP-hard for any fixed $m \ge 2$, even if we have complete information about our problem (e.g. how much time each process will require and how processes will synchronize). In fact, our result is stronger than this since we show the problem remains NP-hard even when $(A, \subset)$ is a set of chains (or a tree) of actions.

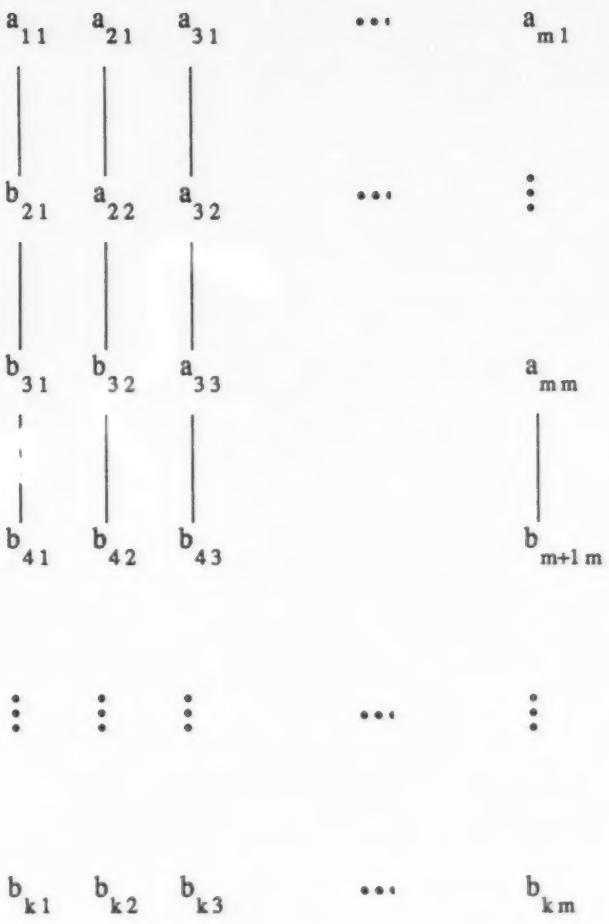

Figure 1a: Partially ordered actions (example 1)

$a_{m\,1}$	$a_{m-1\,1}$	$a_{m-2\,1}$	$\cdots$	$a_{1\,1}$	$b_{2\,1}$	$b_{3\,1}$	$\cdots$	$\cdots$	$b_{k-2\,1}$	$b_{k-1\,1}$	$b_{k\,1}$
	$a_{m\,2}$	$a_{m-1\,2}$	$\cdots$	$a_{2\,2}$	$b_{3\,2}$	$b_{4\,2}$	$\cdots$	$\cdots$	$b_{k-1\,2}$	$b_{k\,2}$	
		$a_{m\,3}$	$\cdots$	$a_{3\,3}$	$b_{4\,3}$	$b_{5\,3}$	$\cdots$	$\cdots$	$b_{k\,3}$		
			$\ddots$	$\vdots$	$\vdots$	$\vdots$		$\iddots$			
				$a_{m\,m}$	$b_{m+1\,m}$	$\cdots$	$b_{k\,m}$				

Figure 1b: An optimal schedule (example 1)

$a_{1\,1}$	$b_{2\,1}$	$\cdots$	$b_{k\,1}$	$a_{2\,1}$	$a_{3\,1}$	$a_{4\,1}$	$\cdots$	$a_{m\,1}$					
					$a_{2\,2}$	$b_{3\,2}$	$\cdots$	$\cdots$	$b_{k\,2}$	$a_{3\,2}$	$a_{4\,2}$	$\cdots$	$a_{m\,2}$
											$a_{3\,3}$	$b_{4\,3}$	$\cdots$

Figure 1c: A depth first schedule (example 1)

The decision problem associated with our scheduling problem is:

OPTIMAL PREASSIGNED SCHEDULE (OPS)

INSTANCE: A number of processors, $m \in Z^+$, a deterministic distributed program on m processors, $(A, \subset, f)$, and a time limit, $L \in Z^+$.

QUESTION: Is there a valid schedule of $(A, \subset, f)$ on m processors of length $\leq L$?

We will prove this decision problem is NP-complete.

Theorem 1 OPS is NP-complete

Proof: The problem is similar to flow shop scheduling [8] and the proof of its NP-completeness is similar though simpler.

The following problem, 3-PARTITION, is known to be strong NP-complete [7], i.e. NP-complete even if unary notation is used to represent the integers. To show OPS is NP-complete, we shown in lemma 1 that 3-PARTITION α OPS and in lemma 2 that OPS$\in$ NP.

3-PARTITION

INSTANCE: A bag of s positive integers, $S = \{a_1, a_2, \ldots, a_s\}$ with $s = 3n$, $\sum_{i=1}^{s} a_i = n\,B$ and $B/4 < a_i < B/2$ for $1 \leq i \leq s$.

QUESTION: Does there exist a partition of S into n bags, $S_1, S_2, \ldots, S_n$ such that $\sum_{a \in S_i} a = B$ for $i = 1, 2, \ldots, n$?

[Note that each such subset must contain precisely three elements.]

Lemma 1 3-PARTITION α OPS

Proof: Given an instance, $I = (a_1, a_2, \ldots, a_{3n}, B)$ of 3-PARTITION, construct $g(I) = ((A, \subset, f), m, L)$, an instance of OPS as follows:

Let $m = 2$. A comprises $3n+1$ chains. For each a_i, we have a chain, c_i, of a_i actions assigned to processor 1 followed by a_i actions assigned to processor 2. We also have a chain, d, of length $2nB$ comprising n repeats of B actions assigned to processor 2 followed by B actions assigned to processor 1. Finally m is 2 and L is $2nB$.

Since 3-PARTITION is strong NP-complete we can assume all integers are represented in unary notation and thus g is a polynomial time transformation. We need to prove $I \in Y_{\text{3-PARTITION}} \Leftrightarrow g(I) \in Y_{\text{OPS}}$.

$\Rightarrow$: If $S_i = \{a_{j_{i1}}, a_{j_{i2}}, a_{j_{i3}}\}$ $(1 \le i \le n)$ is the partition of A satisfying $\sum_{a \in S_i} a = B$ for $1 \le i \le n$, then a valid schedule of A is given in Figure 2. In this schedule S_i^1 denotes the actions assigned to processor 1 in chains $c_{j_{i1}}, c_{j_{i2}}, c_{j_{i3}}$ and S_i^2 denotes the actions assigned to processor 2 in those chains.

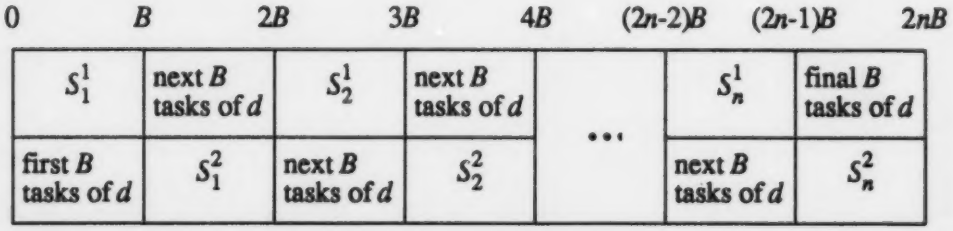

Figure 2: A schedule for OPS

$\Leftarrow$: Since $L = 2nB$ and we have $2 \sum_{a_i \in A} a_i + 2nB = 4nB$ actions allocated to only two processors, the schedule can have no idle time whatsoever. Also since d is a chain of $2nB$ actions the schedule of these actions is fixed as in Figure 2.

Now, consider the actions scheduled on 2 in $[B, 2B[$. They belong to chains $\{c_i\}$ but since there must be B such actions, we must have at least three chains contributing actions. These chains must all have had all their first halves processed on machine 1 in time $[0, B[$. At most three such top halves could have been processed in time $[0, B[$. Hence we deduce that exactly three top halves are processed in $[0, B[$ and their corresponding bottom halves in $[B, 2B[$. We can continue this reasoning and deduce that for any $[(2k\text{-}2)B, (2k\text{-}1)B[$, $1 \leq k \leq n$, the top halves of precisely three chains are processed and in $[(2k\text{-}1)B, 2kB[$ the corresponding bottom halves are processed. Say, $\{c_{j_{k1}}, c_{j_{k2}}, c_{j_{k3}}\}$ are the three chains processed in $[(2k\text{-}2)B, 2kB[$, $1 \leq k \leq n$, then the sets $S_k = \{a_{j_{k1}}, a_{j_{k2}}, a_{j_{k3}}\}$ form a 3-partition of S. The proof of lemma 1 is thus completed.

The proof of theorem 1 then follows from the following result.

Lemma 2 OPS $\in$ NP

Proof: For each processor, k, consider the n_k actions assigned to k and arbitrarily "guess" a (priority) ordering of these actions. Once these priority orderings have been defined, there exists a unique valid schedule respecting these priorities, viz at time t, processor k schedules the highest priority action amongst those not previously scheduled but whose ancestors have been scheduled at time $< t$. A correctly guessed collection of priority orderings will result in an optimal schedule.

The proof of theorem 1 also establishes the following strong result.

Corollary 1.1 OPS is NP-complete even when m is fixed at any value ≥ 2 and $(A, \subset)$ is constrained to be a set of chains.

V. Level Scheduling

A successful heuristic in job shop scheduling, where any processor can perform any action, is the level scheduling strategy [10]. This heuristic is used since finding an optimal schedule is NP-hard for unrestricted job shop scheduling. We will see that this heuristic does not work for our problem.

Using the terminology of this paper, an action $a \in A$ is a terminal action iff there is no $b \in B$ such that $a \subset b$. A chain of length n in A from a to b is a sequence of actions in A, $a = a_0 \subset a_1 \ldots \subset a_{n-1} \subset a_n = b$. The level of $a \in A$ is the length of the longest chain from a to a terminal action. A schedule s of $(A, \subset, f)$ is called a level schedule iff each processor, given a choice of possible actions to process at any time, always selects an action at the greatest possible level.

We have argued that at the time a is scheduled, we cannot know the descendants of a. Hence the level of a can only be finally determined when the whole program is completed. Since level scheduling does not necessarily lead to a good schedule, the fact that we cannot implement it is not a serious problem. On the other hand, if level scheduling did lead to a good, if not optimal schedule, it might suggest the way for more practical heuristics.

Consider the deterministic distributed program illustrated in Figure 3a, which we will refer to as example 2. As before, each subscripted variable represents an action. As before, the second subscript indicates the processor to which an action is assigned. There are m processors, and k is an arbitrary, large, value.

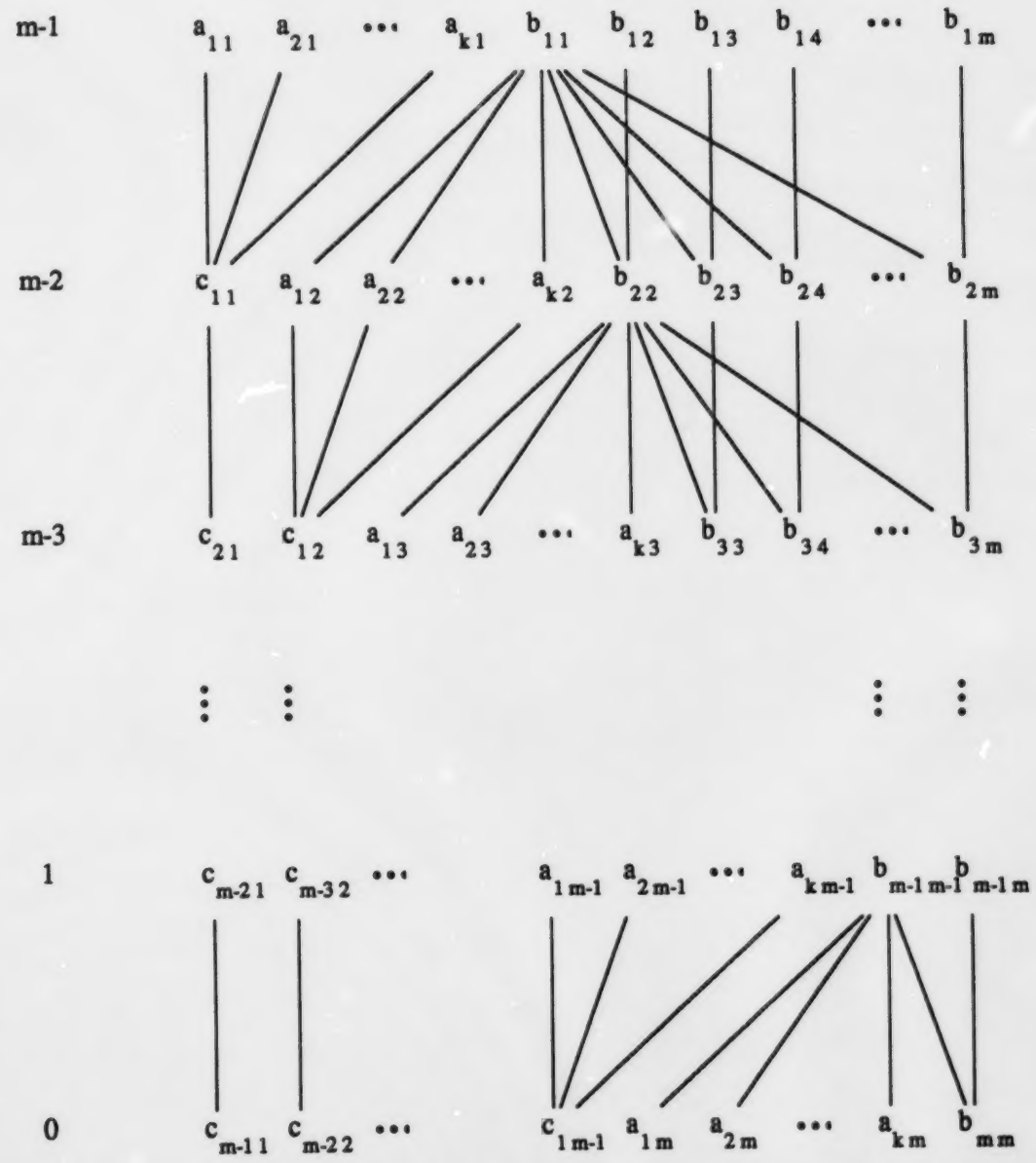
level
m-1
m-2
m-3
1
0
a11 a21 ••• ak1 b11 b12 b13 b14 ••• b1m
c11 a12 a22 ••• ak2 b22 b23 b24 ••• b2m
c21 c12 a13 a23 ••• ak3 b33 b34 ••• b3m
cm-21 cm-32 ••• a1m-1 a2m-1 ••• akm-1 bm-1m-1 bm-1m
cm-11 cm-22 ••• c1m-1 a1m a2m ••• akm bmm

Figure 3a: Partially ordered actions (example 2)

At each level there are $k+m$ actions. Notice how the number of b actions decreases by one and the number of c actions increases by one each time we go down one level. Notice also how each process of the form $b_{i\,i}$ (i.e. each b with equal subscripts) is a critical process, with all actions of type a or b at the next level down having to wait for the critical process to be performed. Whenever we have a critical action, $b_{i\,i}$, we also have k actions of type a assigned to the same processor.

There are a total of $m(m+k)$ actions in the m rows of Figure 3a. Hence, if all actions were moved to a single processor, the length of a schedule would be $m(m+k)$, assuming the processor is not idle when there is work to be done.

Figure 3b gives an optimal schedule for this program on m processors. Whenever a processor has a choice as to which action to perform next, it chooses to do all type b actions before doing any type a action. The length of this schedule is $m+k$, giving a speed-up of m, which is the best possible speed-up on m processors.

On the other hand, Figure 3c contains a level schedule where each processor always performs type a actions before type b actions, when given the choice. For example, processor 1 spends k units of time performing a's and then one unit of time performing b_{11}. During this time, the other processors each perform one type b action and then sit idle for k units of time. Next, processor 2 spends k units of time on a's, followed by on unit of time performing a critical b, while processors 3 through m perform only one more noncritical b and processor 1 finishes up its cs and then sits idle forevermore. The schedule length is $m(k+1)$, assuming $k > m$, giving a speed-up of $\frac{m(m+k)}{m(k+1)}$, which can be made arbitrarily close to 1 by making k large enough. Notice that a processors is idle only when it has no runnable actions.

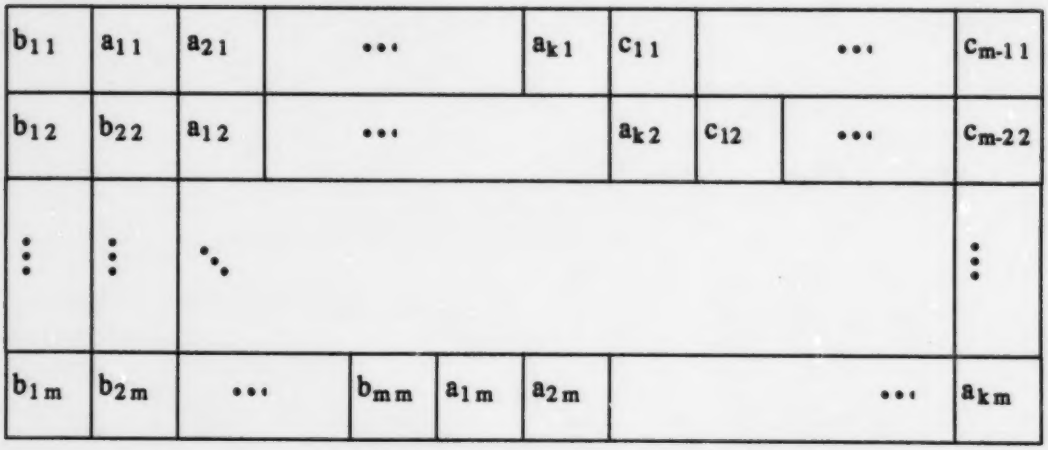

Figure 3b: An optimal schedule (example 2)

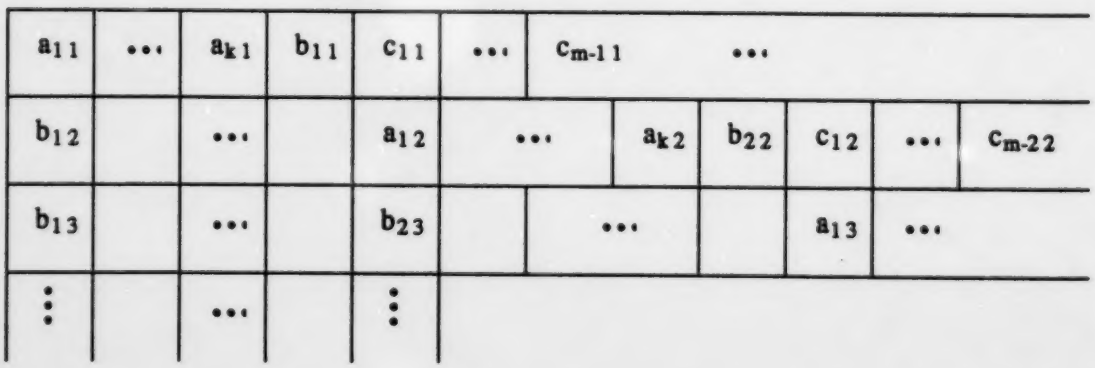

Figure 3c: Worst possible schedule (example 2)

It is worth noting that if scheduling were done in a completely random way (which is the best we can hope for if we cannot distinguish critical actions from other actions), then we would expect to do half of the *a*s before doing the critical *b*, which would improve things by only a factor of 2. That is, our speed-up in this example would approach 2 rather than 1, no matter how large m is.

Level scheduling can thus do as badly as any work preserving scheduling strategy on certain programs. If the program is not distributed, so any processor can perform any action, and $(A, \subset)$ is a tree, then level scheduling produces an optimal job shop schedule. The fact that this result cannot hold for our model follows from corollary 1.1.

VI. Process Placement

So far, we have assumed that all actions are preassigned to specific processors. In this section we will assume that actions are grouped into processes, that all actions in a given process must be assigned to the same processor, but that different processes may be assigned independently. We will show by example that it is not possible to place processes wisely without knowledge of what is yet to happen.

We will remove the restriction that each action requires exactly one unit of time to execute. Consider the collection of m^2 processes, to be run on m processors, illustrated in Figure 4. Each process consists of two actions, an *a* action and a corresponding *b* action. No *b* action can be performed until all *a* actions are finished. We will assume that each *a* action requires one unit of time, but *b* actions are of two types. *Nice* type *b* actions require one unit of time, but *nasty* type *b* actions require k units of time, for some arbitrary, large, value k. (If the reader wishes to continue to think in terms of unit execution time actions, replace each nasty type *b* action with a chain of k unit execution time actions.) A process

containing a nice or nasty action will be called a nice or nasty process, respectively. There are m nasty processes and $m(m-1)$ nice processes.

We will assume that a computer system must assign each process to a processor before that process starts execution. Furthermore, we will assume that the system cannot distinguish a nasty process from a nice one before the type b action starts executing. (Again, if we could tell how much time a computation was going to require in advance, we could solve the halting problem.) Since all type a actions must be performed before any type b action, all processes must be assigned to processors before the character of any process is known.

In the best case, each processor will be assigned 1 nasty process and m-1 nice processes. The resulting schedule will require $2m+k-1$ units of time (2 units for each nice process and $k+1$ units for the nasty process). All processes will be busy until all work is done, so the speed-up will be m.

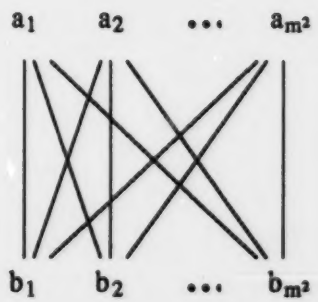

Figure 4: Synchronizing processes

On the other hand, with m^2 processes, at least one processor must have at least m processes, no matter how processes are assigned to processors. If one processor happens to be assigned all m nasty processes, then the schedule length will be $m(k+1)$, giving a speed-up of $\frac{m(2m+k-1)}{m(k+1)}$. As k becomes large, nasty processes become increasingly dominant and the speed-up drops to arbitrarily close to 1. Hence, with process placement, as with scheduling, there are cases where programs may have a speed-up equal to the number of processors, virtually no speed-up, or anything in between. In this case, if processes were assigned to processors in a completely random fashion, we would not expect to do too badly.

The problem of assigning processes to processors is considered in more detail in [3].

VII. Conclusion

We have seen that it is not safe to rely on automatic process placement or scheduling for arbitrary programs in a distributed system where processes cannot be moved. There are several possible ways around this problem.

The most obvious solution is to use a system where processes can be freely moved. We believe that trying to do this in a distributed system will introduce bad worst case communication costs. A more practical approach might be to use a shared memory system, or at least a system where remote memory accesses are affordable.

Even with shared memory parallel computers there are analogous problems. With such architectures and with fine grain parallelism, memory contention becomes the problem. If we view actions as accessing a particular memory module rather than being performed on a

particular processor, then the order in which actions access a memory modules is analogous to the order in which actions are executed on a processor. In fact, the same is true of any critical resource where each action is required to use a particular instance of the resource.

It has been observed that in practice even fairly simple load balancing algorithms tend to give good performance for certain types of distributed computing applications [5]. If we take reasonable care in the type of distributed program we write, and are willing to take our chances with no guaranteed speedup, then we are likely to be in luck most of the time. This approach should not be considered for safety critical real-time programs.

Going a step further, the class of permitted distributed programs can be limited by constraints in the language. However, the first author is interested in parallel and distributed implementations of functional programming languages, which tend to result in highly dynamic programs. We do not have a good characterization of the class of programs that are safe. Since even simple chains are causing significant problems this may be an unpromising avenue of research.

Letting the programmer manage process placement and scheduling appears to be another solution. This may be practical in applications where there are a fixed number of processes, with this number depending on the number of processors. Since the scheduling problem appears to be more critical than the problem of assigning processes to processors, a system might place processes randomly and require a programmer may assign priorities to processes to guide a scheduler.

However, if the goal for a particular computer system is to support general purpose parallel processing at a high level, with dynamic process creation and arbitrary process

synchronization, then we believe that a system allowing cheap process migration is essential for guaranteed performance.

References

[1]. F. W. Burton, "Storage management in virtual tree machines", *IEEE Trans. Comp. 37* (March 1988), 321-328.

[2]. F. W. Burton, G. P. McKeown, V. J. Rayward-Smith and M. R. Sleep, "Parallel processing and combinatorial optimization", *Proc. 1981 Conf. on Combinatorial Optimization*, Stirling, Scotland, Aug. 1981, 19-36.

[3]. F. W. Burton, G. P. McKeown and V. J. Rayward-Smith, "On process assignment in parallel computing", *Info.Processing Letters 29* (15 Sept. 1988), 31-34.

[4]. E. G. Coffman Jr. (Ed). *Computer and Job Shop Scheduling Theory*. John Wiley & Sons, New York, 1976.

[5]. D. L. Eager, E. D. Lazowska and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems", *IEEE Trans. Softw. Eng. SE-12* (May 1986), 662-675.

[6]. D. L. Eager, J. Zahorjan and E. D. Lazowska, "Speed-up versus efficiency in parallel systems", *IEEE Trans. Comput. 38* (March 1989), 408-423.

[7]. M. R. Garey and and D. S. Johnson, *Computers and Intractability: A guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[8]. T. Gonzalez and S. Sahni, "Flowshop and jobshop schedules: complexity and approximation", Operations Research, Vol 26, pp 36-52 (1978)

[9]. T. H. Lai & A. Sprague, "Performance of Parallel Branch-and-Bound Algorithms", *IEEE Trans. Comput. 34* (1985) pp. 962-964.

[10]. S. Lam and R. Sethi, "Worst case analysis of two scheduling algorithms", *SIAM J. Comput. 6* (1977), 518-536.

[11]. J. M. Jaffe, "Bounds on the Scheduling of Typed Task Systems", *SIAM J. Comput. 9* (1980) pp.541-551.

Centre for Systems Science

• Simon Fraser University established the **Centre for Systems Science** as a provincial centre for advanced study. The objective of CSS is twofold: coordinate and combine advanced systems research at SFU, and encourage application of research results within industry in order to further the technological development of British Columbia.

• CSS has assembled three research groups to streamline its efforts. The orientation of these groups is as follows: intelligent systems, microelectronics, and computer and communications systems. Because of shared research interests, this latter group merges with LCCR (**Laboratory for Computer and Communications Research**) for the publication of technical reports.

• CSS's three groups involve over 80 skilled faculty researchers principally from SFU's Schools of Engineering Science and Computing Science, but also draw substantially on Mathematics, Physics, Geography, Philosophy, Linguistics, Psychology, Communication, and Kinesiology.